A communication-avoiding, hybrid-parallel, rank-revealing orthogonalization method

Mark Hoemmen
Scalable Algorithms Department, Sandia National Laboratories
P.O. Box 5800, MS 1320, Albuquerque, NM 87185-1320, USA
Email: mhoemme@sandia.gov

***Abstract*—Orthogonalization consumes much of the run time of many iterative methods for solving sparse linear systems and eigenvalue problems. Commonly used algorithms, such as variants of Gram-Schmidt or Householder QR, have performance dominated by communication. Here, "communication" includes both data movement between the CPU and memory, and messages between processors in parallel. Our Tall Skinny QR (TSQR) family of algorithms requires asymptotically fewer messages between processors and data movement between CPU and memory than typical orthogonalization methods, yet achieves the same accuracy as Householder QR factorization. Furthermore, in block orthogonalizations, TSQR is faster and more accurate than existing approaches for orthogonalizing the vectors within each block ("normalization"). TSQR's rank-revealing capability also makes it useful for detecting deflation in block iterative methods, for which existing approaches sacrifice performance, accuracy, or both.**

We have implemented a version of TSQR that exploits both distributed-memory and shared-memory parallelism, and supports real and complex arithmetic. Our implementation is optimized for the case of orthogonalizing a small number (5–20) of very long vectors. The shared-memory parallel component uses Intel's Threading Building Blocks, though its modular design supports other shared-memory programming models as well, including computation on the GPU. Our implementation achieves speedups of 2 times or more over competing orthogonalizations. It is available now in the development branch of the Trilinos software package, and will be included in the 10.8 release.

***Index Terms*—communication-avoiding; iterative methods; numerical linear algebra; orthogonalization; parallel; QR**

I. Introduction

The Tall Skinny QR factorization (TSQR) is a family of algorithms for computing the QR factorization of a matrix with many more rows than columns. Demmel et al. [1] describe TSQR in detail, and give a full analysis of its communication and computational complexity. Our original motivation for TSQR is to improve the performance and accuracy of the *normalization* step in Block Gram-Schmidt orthogonalization. *Block Gram-Schmidt* (BGS) orthogonalizes entire groups of vectors ("blocks") at a time. For a detailed analysis, see e.g., Stewart [2]. Given a block of vectors X and k blocks of previously orthogonalized basis vectors $Q_1, \ldots, Q_k$, BGS orthogonalizes X in two steps:

- *Project*: For each Q_j, compute the block inner product $C_j := Q_j^* X$ and the update $X := X - Q_j C_j$. When finished, X is orthogonal to $Q_1, \ldots, Q_k$.
- *Normalize*: After the projection step above, orthogonalize the columns of X. Optionally, detect the rank of X and identify a basis for the column space, if X is rank deficient after projection.

Existing normalization algorithms include Classical Gram-Schmidt (CGS), Modified Gram-Schmidt (MGS), the Householder QR factorization, and an approach based on the Cholesky or eigenvalue factorization of the Gram matrix of X described by Stathopoulos and Wu [3]. In Demmel et al. [1], we show that TSQR communicates asymptotically less than CGS, MGS, or Householder QR, and no more than the method of Stathopoulos and Wu. Communication dominates the performance of orthogonalization methods when vector lengths are much longer than the number of vectors, which is the typical case. Furthermore, TSQR is also just as accurate as Householder QR, and more accurate than all the other approaches.

Our original interest in normalization kernels comes from our Communication-Avoiding GMRES (CA-GMRES) algorithm, which we summarize in Demmel et al. [4] and describe in detail (along with other communication-avoiding iterative methods) in Hoemmen [5]. CA-GMRES is a rearrangement of standard GMRES that computes vectors in blocks, rather than one at a time. One step of CA-GMRES works like this:

1) Generate a small number s of candidate Krylov basis vectors, using an efficient computational kernel called the "matrix powers kernel" (see also [6], [7]).
2) *Project* the s candidate vectors against previously orthogonalized blocks of basis vectors.
3) *Normalize*: Use TSQR to make the s vectors mutually orthogonal.

In CA-GMRES, the number of columns s per block is typically small – perhaps only 5 or 10 (see Demmel et al. [4]). *Block iterative methods*, for solving clustered eigenvalue problems and linear systems with multiple right-hand sides, are another set of algorithms which can benefit from better normalization. See e.g., Golub and Underwood [8], O'Leary [9], Stewart [2], and Baker et al. [10]. Block iterative methods also use Block Gram-Schmidt. The application often dictates the number of vectors per block, but it is common to use blocks of no more than 20 vectors.

The small number of columns in the block to normalize affects TSQR optimizations in ways incompatible with existing implementations of TSQR, such as in the PLASMA and MAGMA projects (see Agullo et al. [11]). These implementa-

tions use TSQR as the panel factorization in a general dense QR factorization, called CAQR (see Demmel et al. [1]). CAQR requires dividing the panel into square blocks. If the number of columns in the panel is too small, this introduces too much overhead. As a result, TSQR implementations intended for this application are optimized for the case of square blocks, of a size large enough to justify the overhead of reorganizing data into cache blocks and using dynamic scheduling. In contrast, our implementation assumes blocks with more rows than columns, in order to use cache more effectively and minimize overhead.

Many Krylov and block Krylov methods for sparse eigenvalue problems require a rank-revealing orthogonalization method. TSQR can be easily adapted for this purpose, and we have done so in our implementation. It is potentially more accurate and faster than implementations based on Gram-Schmidt orthogonalization, such as the Modified Gram-Schmidt reorthogonalization procedure with randomization described by Stewart [12]. (See Hoemmen [5, Chapter 2] for a fuller discussion.) There are many possible versions of rank-revealing TSQR, but they all begin by computing the QR factorization $A = QR$. Since R is much smaller than A, computations on R are practically free. For example, we can compute a QR factorization with column pivoting:

$$A = QR, R\Pi = Q_R R_R, A\Pi = (QQ_R)R_R,$$

a singular value decomposition (SVD):

$$A = QR = Q(U\Sigma V^*) = (QU)\Sigma V^*,$$

or a polar decomposition:

$$A = QR = Q(U\Sigma V^*) = (Q(UV^*))V\Sigma V^*.$$

The polar decomposition is an essential tool for solving the orthogonal Procrustes problem (Higham [13]). It may be useful for Krylov methods because the orthogonal[1] polar factor $Z = Q(UV^*)$ is unique in exact arithmetic. This is not necessarily true of the orthogonal factor Q in a QR factorization. It is only unique up to a unitary scaling of the columns of Q and the rows of R. For example, this means that the diagonal of R may have negative entries. However, Krylov methods like GMRES and Arnoldi require the diagonal of the R factor to be nonnegative (see [5, Chapter 3] for details). In Demmel et al. [14], we explain how to change how the Householder reflectors in the QR factorization are computed, so that the R factor always has a nonnegative diagonal. This also makes Q unique in exact arithmetic. This capability has been in LAPACK since version 3.2. However, as of LAPACK version 3.2.2, this feature is not enabled by default for the QR factorization, since some inputs exist for which it results in somewhat degraded accuracy. This does not appear to affect the accuracy of TSQR in practice, as we show experimentally in Section V. However, using TSQR to compute a polar factorization solves this problem, with appropriate modification of GMRES and Arnoldi so that they work with a block upper Hessenberg matrix (such as in the Block Krylov-Schur method [2]).

A third class of applications for TSQR involves matrices with many more rows than columns, but where the number of columns is large enough that the approach of this paper may not suffice for good performance. Many data analysis and distillation applications fall into this category. For example, Robust Principle Component Analysis (see Candès et al. [15]) involves computing the SVD of matrices with perhaps millions of rows, but only a hundred or so columns. The number of floating-point operations in both TSQR and ordinary Householder QR increases with the square of the number of columns. Thus, the large number of columns increases the cost of floating-point arithmetic relative to data movement costs. We describe in [1] BLAS 3 optimizations that can improve the performance of TSQR in this case, but as the number of columns increases, it may be better to use a more general QR factorization. We do not address this class of applications in this paper.

This paper is organized as follows. Section II summarizes the algorithms underlying TSQR, which the full implementation composes hierarchically to exploit different levels of parallelism effectively. Section III explains the computational kernels in which TSQR performs all its floating-point operations. We show performance results for TSQR's components in Section IV, and summarize accuracy results in Section V. Section VI gives performance results for the full TSQR implementation, both alone and combined with block orthogonalization. We conclude and discuss ongoing work in Section VII.

II. Algorithm

In this section, we summarize the TSQR family of algorithms, and explain its various components. In [1], we describe both parallel and sequential cache-blocked versions of TSQR in detail. We construct a performance model including arithmetic and communication, and compare this model with models of Modified Gram-Schmidt and Householder QR, showing that parallel TSQR requires less communication between processors, and sequential cache-blocked TSQR requires less data movement between cache and main memory. In [4], we discuss a shared-memory parallel, cache-blocked implementation of TSQR, as one component in a prototype of a new iterative solver.

This work combines these results to exploit two levels of parallelism:

- Distributed-memory, via the Message Passing Interface (MPI) [16]
- Shared-memory, currently via Intel's Threading Building Blocks [17] framework (though the modular structure of the library makes it easy to swap in other implementations)

The resulting TSQR library works for the same real and complex data types supported by the BLAS and LAPACK. In this paper, when we refer to the "full TSQR algorithm" or to

[1]We use the term "orthogonal" to mean both orthogonal in real arithmetic, and unitary in complex arithmetic.

"TSQR" without further qualification, we mean this two-level parallel approach.

A. The full TSQR algorithm

1) Factor phase: TSQR begins with an $m \times n$ matrix A with $m \gg n$, stored in block row fashion among P MPI processes. Each process k has an $m_k \times n$ block A_k with $m_k \geq n$. First to operate on the matrix is the shared-memory part of TSQR, which must implement an interface we call `NodeTsqr`. Currently, `NodeTsqr` has two different concrete implementations:

- `SequentialTsqr`, a sequential cache-blocked algorithm, suitable for MPI-only parallelism (one MPI process per CPU core)
- `TbbTsqr`, a multithreaded parallel implementation, suitable for hybrid CPU parallelism (multiple CPU cores per MPI process)

`NodeTsqr` on MPI rank k factors that rank's block A_k in place, storing the Q factor implicitly. (The implicit representation depends on the implementation; see Sections II-B and II-D for specific examples.) Then, the distributed-memory part of TSQR, `DistTsqr`, finishes the factorization phase. `DistTsqr` computes the QR factorization of the "R stack": an $nP \times n$ matrix distributed among P MPI processes, where each rank k's part of the matrix is an $n \times n$ upper triangular matrix R_k. There are two different versions of `DistTsqr`, which we describe in Section II-C. Both require $\lceil \log_2 P \rceil$ messages on P MPI processes for the factorization phase. The results of this phase are the R factor of the R stack, and a Q factor of the R stack, stored implicitly in distributed fashion among the MPI processes.

2) Explicit Q phase: When the factorization phase is complete, `DistTsqr` then computes the explicit version of the explicitly stored Q factor of the R stack. Since TSQR combines blocks of rows, it cannot form the explicit Q factor in place by overwriting the implicit Q factor, unlike ScaLAPACK's `P_ORGQR` routine.[2] Thus, `DistTsqr` computes the explicit Q factor of the R stack by applying the implicit Q factor to an $nP \times n$ matrix C, consisting of the first n columns of the identity matrix. Each MPI process k has an $n \times n$ block C_k of C. This "apply" phase of the computation again requires $\lceil \log_2 P \rceil$ messages. See Section II-C for diagrams of the factor and apply phases, for both implementations of `DistTsqr`.

After `DistTsqr` has finished its explicit Q phase, each MPI rank k has a copy of the final R factor of the entire matrix A. Then, each MPI process k writes the resulting $n \times n$ matrix C_k into the top block of an $m_k \times n$ matrix Q_k, which initially contains zeros. Finally, on each MPI process k, `NodeTsqr` applies its implicitly stored Q factor to Q_k, resulting in process k's component Q_k of the explicit Q factor. At this point, `NodeTsqr` may either keep its implicitly stored Q factor, or recycle the storage (which is what we generally do when orthogonalizing vectors.)

[2]Throughout this paper, when we refer to ScaLAPACK, LAPACK, or BLAS routines without concern for the specific floating-point data type, we replace the letter in the name corresponding to the data type with an underscore: for example, `_GEMM` instead of `SGEMM`, `DGEMM`, `CGEMM`, or `ZGEMM`.

B. Sequential cache-blocked TSQR

`SequentialTsqr` implements the sequential cache-blocked TSQR algorithm we describe in Demmel et al. [1]. It reads and writes asymptotically less data between cache and main memory than comparable algorithms, such as LAPACK's Householder QR implementation or Modified Gram-Schmidt orthogonalization. Memory traffic is expensive on modern multicore CPUs, so `SequentialTsqr` should run significantly faster than competing algorithms. This component implements the `NodeTsqr` interface described above. It also is invoked by the shared-memory parallel part of TSQR (see Section II-D).

1) Algorithm: `SequentialTsqr` takes as input an $m \times n$ input matrix A stored in column-major order. It divides up the matrix by row blocks into cache blocks. The cache blocks have the same stride as A, so the caller's data does not need to be reorganized. (There is also an option to reorganize cache blocks into contiguous storage, which we discuss below.) `SequentialTsqr` then makes two passes over the matrix:

1) A "factor phase," which passes from top to bottom sequentially through the cache blocks of A. It computes the R factor, and overwrites the input matrix with an implicit representation of the Q factor.
2) An "apply / explicit Q phase," which passes from bottom to top (when applying Q; when applying Q^T or Q^H, top to bottom) sequentially through the cache blocks of A. It applies the implicit Q factor to the first n columns of the identity matrix, thus computing the explicit Q factor. (TSQR cannot compute the Q factor in place; there is no equivalent to LAPACK's `_ORGQR` routine.)

`SequentialTsqr` can function as a standalone QR factorization. (With the right interface, it would be a drop-in replacement for LAPACK's `_GEQR2` and `_ORMR2` panel QR factorization routines.) However, when using `SequentialTsqr` in combination with other TSQR components, such as `DistTsqr` (see Section II-C) and / or `TbbTsqr` (see Section II-D), the apply phase of `SequentialTsqr` involves applying the implicitly stored Q factor to an $m \times n$ matrix consisting of all zeros except for the top $n \times n$ top block. That $n \times n$ block in turn comes from the apply phase of `DistTsqr` resp. `TbbTsqr`.

2) Tuning parameters: `SequentialTsqr` has two tuning parameters: the cache size in bytes, and whether cache blocks are stored contiguously. The cache size setting bounds above the number of rows in each cache block in the matrix. We currently allow users to set the cache size themselves, and pick a reasonable default if they do not. Discovering it automatically is future work. If the cache size is set too large, TSQR will thrash the cache. Our implementation runs one instance of sequential cache-blocking TSQR per CPU core, so for best locality, it makes sense to choose the cache size as that of the largest-level private cache for each core. If cores

only have a small private L1 and a shared L2, then it makes sense to divide the L2 size by the number of cores sharing it.

The second tuning parameter governs whether TSQR assumes that the matrix is stored with contiguous cache blocks, rather than in the usual column-major order. We provide routines for converting to and from this layout. Its advantage should be to reduce the total number of memory pages loaded for very large matrices, thus avoiding TLB (Translation Lookaside Buffer) thrashing. However, our performance experiments (see Section IV-C) show that this layout does not significantly improve performance for the matrix sizes we tested.

C. Distributed-memory parallel TSQR

The `DistTsqr` module performs all communication between MPI processes in TSQR. We implemented two different algorithms for `DistTsqr`: "Butterfly," and "Reduce and Broadcast."

In "Butterfly," pairs of MPI processes exchange data at each stage, as in an all-reduce. We implement the exchange using a nonblocking send and receive between each pair of processes. The factor phase, with a critical path length of $\lceil \log_2 P \rceil$ on P processes, computes the R factor (which ends up on all processes) and an implicit representation of the Q factor of the R stack. Both are computed redundantly. Figure 1 illustrates the factor phase of Butterfly. The apply phase, shown in Figure 2, applies the implicit Q factor of the R stack to a matrix C. This phase also has a critical path length of $\lceil \log_2 P \rceil$ on P MPI processes. If the matrix C contains n columns of the identity, then the explicit Q factor of the R stack is computed. This involves unnecessary work, since many processes end up exchanging and operating on zeros. However, the Butterfly version can also apply the Q factor (or Q^T or Q^H, with a reversed version of the algorithm) to an arbitrary matrix (e.g., when solving least-squares problems). The Reduce and Broadcast version cannot do this.

The second algorithm, called "Reduce and Broadcast" (RB), uses a different communication pattern. In the factor phase, shown in Figure 3, only one process in a pair sends, and the other process receives, computes, and stores the results. The resulting R factor of the R stack ends up on MPI rank 0, as in a reduction to rank 0. The implicit Q factor is stored in a tree, without redundancy. The second "explicit Q" phase, illustrated in Figure 4, reverses this process, as in a broadcast. RB's second phase is only capable of computing the explicit Q factor, since it applies the Q factor to a matrix distributed among processes, where only the root process' part of the matrix has nonzero entries. This is the typical use case of TSQR in Krylov subspace methods. However, users must commit to this representation when they first factor the matrix; they cannot use the results to apply Q to a general matrix.

The disadvantage of the Butterfly implementation is that it sends and receives many more messages at once. In particular, Reduce and Broadcast only sends or receives at most one message per multicore node at a time. In contrast, at each stage of the factorization and apply / explicit Q phases, Butterfly both sends and receives as many messages per node as there

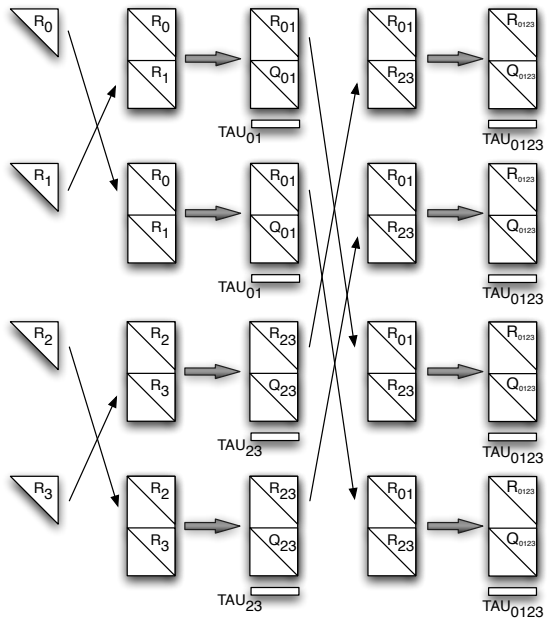

Fig. 1. Factor phase of Butterfly implementation of `DistTsqr`, on $P = 4$ MPI processes, of an $nP \times n$ matrix. See the caption of Figure 3 for a key.

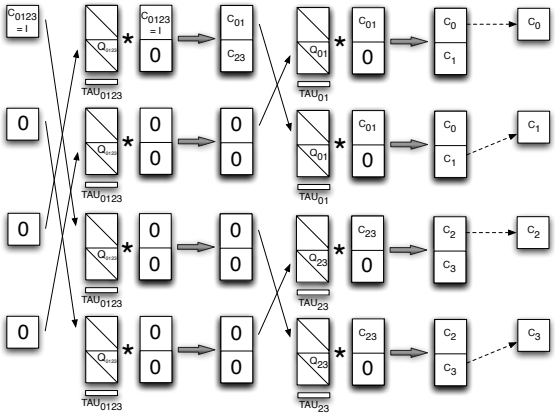

Fig. 2. Apply / Explicit Q phase of Butterfly implementation of `DistTsqr`, on $P = 4$ MPI processes, of an $nP \times n$ matrix. See the captions of Figures 3 and 4 for a key. Note that this part of the algorithm sends, receives, and computes on zeros, when computing the explicit Q factor. When applying Q to a matrix, those zeros are replaced with significant data. The Reduce and Broadcast implementation can only compute the explicit Q factor; it cannot apply Q to a matrix.

are MPI processes on that node. If messages serialize at the network interface of the node, this can result in a bottleneck. We expect MPI process counts per node to increase, given that that number of NUMA (Non-Uniform Memory Access) regions is increasing, and running one MPI process per NUMA region offers best performance overall for solvers (see Edwards [18]). Thus, we expect the Reduce and Broadcast variant to be faster, especially for large numbers of MPI processes. Indeed, we found this to be the case, as our performance results in Section IV-D show.

We were surprised, however, that the Butterfly variant

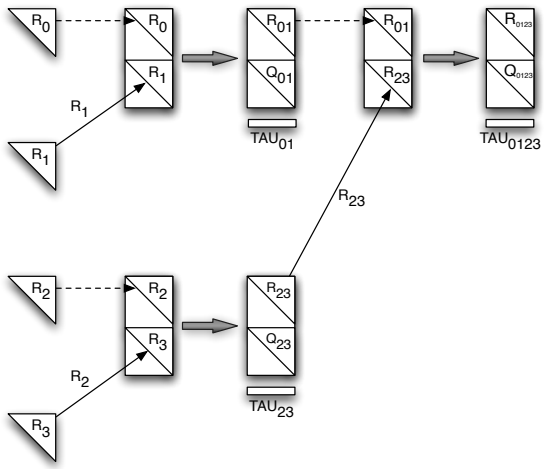

Fig. 3. Factor phase of Reduce and Broadcast implementation of `DistTsqr`, on $P = 4$ MPI processes, of an $nP \times n$ matrix. Time increases from left to right. Initially, MPI rank k owns R_k. Solid black arrows indicate a message (whose contents are shown next to the array), and dotted black arrows show which data doesn't need to be sent or received at a particular step. Grey thick arrows indicate a computation (in this case, the QR factorization of two vertically stacked R factors). The "TAU" arrays are each of length n.

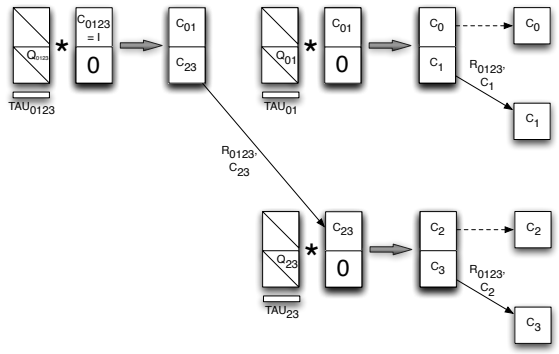

Fig. 4. Explicit Q phase of Reduce and Broadcast implementation of `DistTsqr`, on $P = 4$ MPI processes, of an $nP \times n$ matrix. See the caption of Figure 3 for a key. Also, the asterisks indicate applying an implicitly stored intermediate Q factor to a matrix.

performed better for small MPI process counts. This may have something to do with our use of nonblocking communication in that implementation, even though neither algorithm offers an opportunity to overlap communication and computation. Future work may include automatic run-time performance tuning to pick the fastest implementation of `DistTsqr`. The advantage of Butterfly is that it works both for computing the explicit Q factor, and for applying Q, Q^T, or Q^H to an arbitrary matrix.

Both implementations of `DistTsqr` use a binary tree, though they accept arbitrary MPI process counts. If the number of processes is not a power of two, some of the processes idle at some stages. A binary tree is not necessary; any tree shape would work, as long as the factor and apply / explicit Q parts uses the same tree shape. (This is because `DistTsqr` stores data in the interior nodes of the tree. As a result, `DistTsqr` cannot be implemented using MPI's reduce or all-reduce collective operations.) We may investigate other tree shapes in future work.

D. Shared-memory parallel TSQR

The `TbbTsqr` module implements a shared-memory parallel version of TSQR within a single MPI process. It thus implements the `NodeTsqr` interface. The parallel part of `TbbTsqr` works much like the Butterfly version of `DistTsqr` (see Section II-C), except that since it runs in a single memory space, it does not need to send messages between processors. `TbbTsqr` first breaks up the input matrix into a number of row blocks equal to the number of threads to use. It then assigns one `SequentialTsqr` task per thread, which produces an R factor. `TbbTsqr` then combines these R factors in a binary tree, just like `DistTsqr`.

This component uses Intel's Threading Building Blocks framework for parallelism. In particular, it invokes the `tbb::task` interface directly to assign tasks to the dynamic scheduler. We set as many tasks as there are CPU cores (or let the user specify how many tasks to use). Each task first performs sequential cache-blocked TSQR on its part of the matrix, and then the tasks combine their results in a binary tree. This approach minimizes the number of synchronization points between threads. While it does nothing to optimize for Non-Uniform Memory Access (NUMA), we consider NUMA optimizations at the shared-memory level of TSQR unnecessary. This is because for iterative solvers, the best overall configuration of MPI and shared-memory parallelism is to run one MPI process per NUMA region, and parallelize with threads inside each NUMA region (see Edwards [18]). With the appropriate settings, each MPI process will stay within its NUMA region, and it will only allocate memory within that reason. In contrast, typical shared-memory programming models require much more programmer effort in order to ensure that each thread only works on data within its NUMA region (see e.g., [4]).

III. Computational kernels

In this section, we describe the `Combine` module. This module implements the computational kernels in which TSQR performs all its floating-point arithmetic. The module's name indicates that the kernels *combine* the results of previous steps in the factorization. For good performance, these kernels must operate on the data in place, rather than copying in and out of temporary storage. TSQR works in a memory bandwidth - limited regime, so it cannot afford to do more copying than necessary.

In Section III-A, we describe the four kernels and show where they are used by higher-level TSQR components. In Section III-B, we explain our optimization choices. Section IV-B will compare the performance of these kernels with that of LAPACK's corresponding kernels.

A. The four kernels

The four kernels come in "factor" and "apply" pairs. "Factor" kernels correspond to LAPACK's _GEQRF routine (QR factorization, overwriting the input with the R factor and an implicit representation of Q). "Apply" kernels correspond to LAPACK's _ORMQR (apply the implicitly stored Q factor to a matrix). TSQR does not need an equivalent to LAPACK's _ORGQR, because the row block tree structure of TSQR makes it impossible to compute the explicit Q factor in place.

1) "Inner" kernels: The "inner" kernels work with an input matrix $[R; A]$, where R is $n \times n$ upper triangular, and A is $m \times n$. The matrix A is stored separately from R with a possibly different stride. These kernels are used by the sequential cache-blocked TSQR module (see Section II-B), and they perform most of the floating-point arithmetic in a TSQR invocation. The name "inner" comes from the factorization kernel being an inner, i.e., not the first, step of the sequential cache-blocked TSQR factorization.

The first "inner" routines, `factorInner`, computes the QR factorization of $[R; A]$. It overwrites R with the resulting R factor, and overwrites A with the Householder reflectors implicitly representing the Q factor. Here, A corresponds to a cache block of the input matrix to factor. The `applyInner` routine, in turn, applies the implicitly stored Q factor (or Q^T or Q^H) from `factorInner` to the matrix $[C_1; C_2]$, where C_1 is $n \times q$ and C_2 is $m \times q$, $q \geq 1$, and C_1 and C_2 may have different strides. The typical case is $q = n$. Here, C_2 corresponds to a cache block of the output Q factor.

2) "Pair" kernels: The "pair" kernels work with an input matrix $[R_1; R_2]$, where R_1 and R_2 are each $n \times n$ upper triangular, stored separately with possibly different strides. These kernels are used in both the shared-memory parallel (Section II-D) and distributed-memory parallel (Section II-C) TSQR components.

The `factorPair` routine computes the QR factorization of $[R_1; R_2]$. It overwrites R_1 with the resulting R factor, and overwrites R_2 with the Householder reflectors implicitly representing the Q factor. The `applyPair` routine applies the implicitly stored Q factor (or Q^T or Q^H) from `factorPair` to the matrix $[C_1; C_2]$, where C_1 and C_2 are each $n \times q$ with $q \geq 1$, are stored separately, and may have different strides. The typical case is $q = n$.

B. Optimization choices for the `Combine` kernels

1) In-place operation: Most of the memory traffic in TSQR happens in the "inner" routines. This is because the number of columns n in the matrix is small, but the cache blocks are chosen with enough rows to fill the largest private cache (or portion of a shared cache) belonging to a single CPU core. While one could implement these kernels by copying the input matrices into contiguous storage and calling the corresponding LAPACK routines, we found in previous work (see Demmel et al. [4]) that the overhead of copying the data in and out can double the runtime. The number of columns n is small enough that memory bandwidth is an important part of TSQR's run time. As a result, our implementations work in place on the input data. This feature improves performance, as we show in Section IV-B by comparison with a copy in/out LAPACK implementation (`CombineDefault`).

2) Sequential only: Our implementations of the `Combine` kernels do not exploit shared-memory parallelism. This is because a parallel implementation would require more global synchronization between threads, and would also reduce potential locality. We choose instead a coarser-grained parallelization (over groups of cache blocks) at a higher level than the `Combine` kernels, in order to reduce the frequency of synchronization. However, for graphics processing units (GPUs) or hardware that supports a similar model of parallelism, with fast synchronization among threads in a group, it may make sense to parallelize the `Combine` kernels. Our collaborator, Michael Anderson (see Section VII), has used this technique in his GPU implementation of TSQR. Another way to parallelize the `Combine` kernels would be to use a multithreaded BLAS implementation. We show in Section IV-B that multithreaded BLAS implementations do not parallelize QR effectively when the number of columns is small. Thus, shared-memory parallel TSQR is always a better choice than a parallel `Combine` implementation on multicore CPUs.

3) Fortran vs. C++: Fortran compilers have a reputation for producing faster code than C++ compilers, in part because the Fortran standard allows compilers to assume that arrays are not aliased. We tested this by writing both a C++ and a Fortran implementation (with a thin C++ wrapper) of `TSQR::Combine`: `CombineNative` resp. `CombineFortran`. The results of our performance experiments in Section IV-B show that the Fortran implementation is always faster.

IV. Performance results

In this section, we present TSQR performance results. We first break TSQR into its constituent components, and measure the performance of these separately, compared with competing algorithms. This is legitimate because TSQR operates in phases, so if we take the maximum time over all the MPI processes for each phase, we should get a good indication of the run time of all of TSQR.

Section IV-A discusses the hardware and software configuration used for our experiments. Section IV-B shows performance results for the four computational kernels in `TSQR::Combine`, Section IV-C for sequential cache-blocked TSQR and shared-memory parallel TSQR, and Section IV-D for distributed-memory parallel TSQR. In Section VI that follows, we show representative benchmark results for the full TSQR implementation, composed of all these components.

A. Hardware and software configuration

We ran our performance experiments on a Linux cluster named "Glory," which was built by Appro and funded by the U.S. Department of Energy / National Nuclear Security Administration (DOE/NNSA). Each node of Glory has 16 AMD processors with a clock frequency of 2.2 GHz, arranged in 4 sockets with 4 cores per socket, and 32 GB of DDR2

DRAM. The nodes are connected by 4x Infiniband with an OFED stack (Mellanox ConnectX HCA) and Voltaire DDR Infiniband switches. Glory users may run batch jobs using any number of nodes from 1 node (16 CPUs) to 64 (1024 CPUs).

Most of our performance experiments use the preferred software configuration: Version 11.1 of the Intel C++ and Fortran compilers, OpenMPI version 1.4.1 [19], and Version 11.1 of Intel's Math Kernel Library (MKL). The MKL includes implementations of the BLAS, LAPACK, and ScaLAPACK. To check the effects of BLAS and LAPACK implementation, we ran some experiments with the AMD Core Math Library (ACML) version 4.3. To check the effects of MPI implementation on performance results, we also ran some experiments with MVAPICH version 1.1 [20].

Our benchmarks measure run time for kernels of interest using the MPI-standard `MPI_Wtime` routine, which we measured to have a resolution of about 10^{-6} seconds. We repeat each kernel invocation at least 30 times for slower kernels, and 100 times for faster kernels, and report the mean time. For MPI-parallel benchmarks, we collect timings on all MPI processes, and report the maximum (so, time for the last MPI process to complete). Unless otherwise specified, we show performance results only for the double-precision real floating-point type. Results for other real and complex floating-point types are comparable.

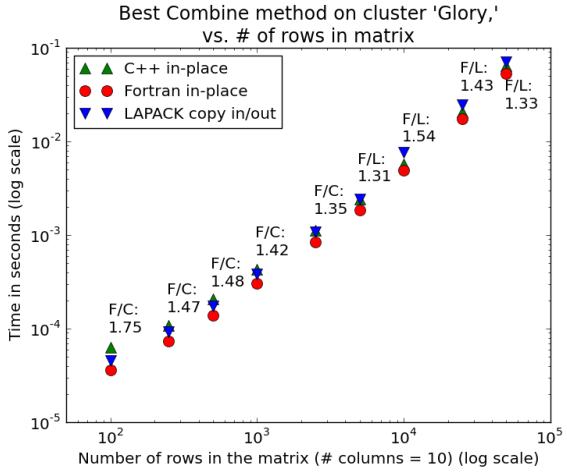

Fig. 5. Run time in seconds of three implementations of `Combine` (`factorInner` and `applyInner` in sequence), on the Glory cluster, for test matrices with $n = 10$ columns and varying numbers of rows. Both axes use a log scale. Green upward-pointing triangles show the C++ implementation, red circles the Fortran implementation, and blue downward-pointing triangles the copy in/out implementation (that invokes LAPACK's `_GEQRF` and `_ORGQR` in sequence, after copying the input matrices into a contiguous buffer). For each number of rows, we show abbreviations for the fastest vs. slowest implementation: "F" for Fortran, "C" for C++, and "L" for LAPACK copy in/out. We also show the speedup resulting from using the fastest implementation vs. the slowest.

B. Performance of fundamental kernels

In Figure 5, we show timings for the "inner" computational kernels in `Combine`, which take up most of the run time of TSQR. In this figure, we compare the three different implementations of `Combine`. We found that the Fortran in-place implementation is always fastest, from 1.31 to 1.51 times faster than the slowest implementation. For smaller matrices, the C++ implementation is slightly slower than copying in and out and using LAPACK. This is to be expected, since we did not optimize the C++ version very much, and it only calls BLAS 1 routines (vs. the LAPACK implementation, which spends most of its time in BLAS 2 routines). However, for larger matrices, the cost of copying in and out makes the C++ in-place implementation faster. In conclusion, we found that computing in place, which is the chief advantage of these custom kernels, significantly improves the performance of `factorInner` and `applyInner`.

We also experimented with using a multithreaded BLAS, but found that using more than one thread actually made the routines slower, for all implementations of `Combine`. It seems that multithreaded BLAS libraries only achieve speedups from BLAS 3 trailing matrix updates. In our case, the number of columns is small enough that not even LAPACK's QR factorization would invoke the BLAS 3 path.

All test matrices have column stride equal to the number of rows. This differs from the typical use case for sequential cache-blocked TSQR and shared-memory parallel TSQR. However, we show in Section IV-C that a longer column stride (as would be the case when working on cache blocks in place, without reorganizing the data so cache blocks are stored contiguously) does not significantly affect performance. Thus, the results of this section reflect performance of the kernels "in place," without reorganization of cache blocks into contiguous storage.

C. Cache-blocked and shared-memory parallel performance

In this section, we report performance results for the sequential cache-blocked part of TSQR, called `SequentialTsqr`, as well as for the shared-memory parallel part of TSQR, called `TbbTsqr`. We can discuss their results together, since `TbbTsqr` for one thread just calls `SequentialTsqr`. Figure 6 shows timings on one node of the Glory cluster, for a 1,000,000 by 10 real double-precision test matrix (factor and explicit Q computation).

Our results show that rearranging the input data so that cache blocks are stored contiguously does not significantly affect performance. This is the case even though we did not measure the time for reorganizing the data. (In any case, this should only be done twice: at startup and at output of final results.) This means that users do not need to pay the performance and usability cost of reorganizing their data. We also found that `TbbTsqr` scales nearly perfectly within a NUMA region (1–4 threads, on the Glory cluster). Outside of the NUMA region (8–16 threads), it does not scale so well. Our shared-memory only benchmark allocates memory before threads begin their work, and it makes no attempt to control NUMA placement. Finally, we found that choosing the cache block size setting near but not greater than each CPU core's cache capacity (in this case, 512 KB) results in generally good performance. Choosing the cache block size too small results

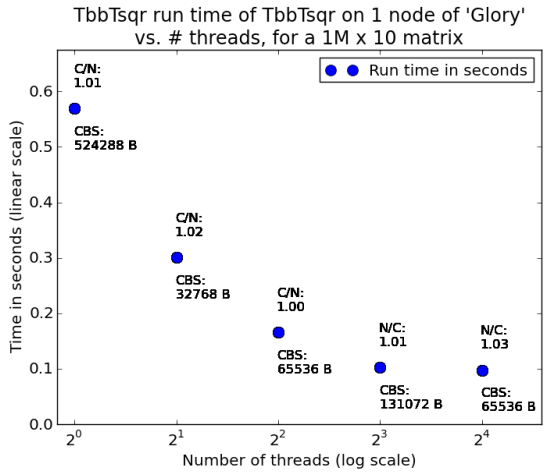

Fig. 6. Run time in seconds of `TbbTsqr` (factor and explicit Q), on one node of the Glory cluster, for a 1M by 10 real double-precision test matrix. Each node of Glory has 16 CPU cores, with 4 cores per NUMA region. Blue circles show results for each number of threads, from 1 to 16. Above each data point is an abbreviation "C/N" or "N/C": if the first letter is "C", then using contiguous cache blocks was faster, else leaving the matrix in its original column-major order was faster. Immediately below the abbreviation is the speedup resulting from the best choice of contiguous vs. noncontiguous cache blocks. Below each data point is the cache block size ("CBS") setting per thread, corresponding to the fastest time for that number of threads. We tested cache block sizes in powers of two from 2^{13} bytes to 2^{20} bytes.

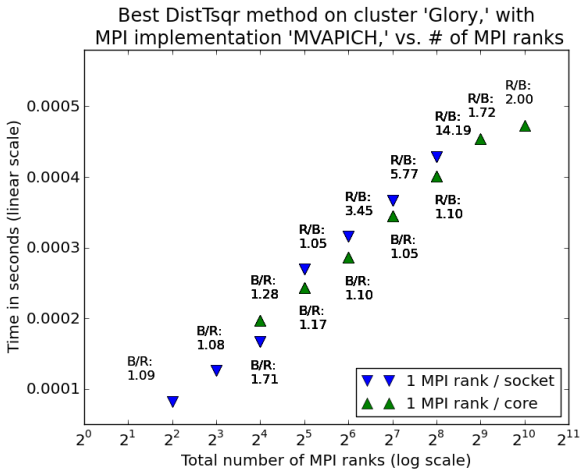

Fig. 7. Run time in seconds of `DistTsqr` on the Glory cluster, with MVAPICH 1.1 as the MPI implementation, for an "R stack" test problem with $n = 10$ columns, on 4–1024 MPI processes. Blue downward-pointing triangles show results for 1 MPI rank per socket (4 ranks per node), and green upward-pointing triangles show results for 1 MPI rank per CPU core (16 ranks per node). (Limits on the number of nodes users are allowed to reserve on this cluster mean that we could not collect 1 MPI rank per socket data for the rightmost two data points.) Each data point is for the best `DistTsqr` implementation, either Reduce and Broadcast (R) or Butterfly (B). Above / below each data point is either B/R or R/B: the first letter indicates the faster `DistTsqr` implementation for that number of MPI processes and number of processes per node, and the number below it is the speedup resulting from using the faster implementation.

in excessive overhead.

We also compared the performance of `TbbTsqr` with LAPACK's QR factorization and explicit Q routine (`DGEQR2` resp. `DORGQR`). As before, we found that using a multithreaded BLAS with more than one thread made the code slower. For the same double-precision real 1,000,000 by 10 test problem, LAPACK required 1.01298 seconds, nearly twice as much time as `TbbTsqr` with one thread, and nearly 10 times as much time as the best multithreaded `TbbTsqr` result.

D. Distributed-memory parallel performance

In Figures 7 and 8, we show timings for the distributed-memory part of TSQR, `DistTsqr`. These benchmarks were performed on the Glory cluster, using two different MPI implementations, on an "R stack" test problem with $n = 10$ columns and nP rows, for various numbers of MPI processes P from 4 to 1024. We found that the Reduce and Broadcast variant of `DistTsqr` outperformed the Butterfly version for large MPI process counts, as expected. Speedups varied, but Reduce and Broadcast was the better choice for 64 or more MPI processes. However, Butterfly outperformed Reduce and Broadcast for less than or equal to 16 MPI processes. Timings varied slightly depending on the MPI implementation, but overall `DistTsqr` performs well.

We also timed ScaLAPACK's QR factorization routines `PDGEQRF` and `PDORGQR`, called as would be done to compute both the R factor and the explicit Q factor (just as `DistTsqr` does). As expected, ScaLAPACK was significantly slower than `DistTsqr`, in fact, overall about 20 times slower. (See Figure 9, and note the difference in scales from the previous two figures.)

We only show performance results for the two cases of one MPI process per NUMA region, and one MPI process per CPU core. Recent experiments by Edwards [18] show that for iterative solvers, running one MPI process per NUMA region results in the best overall performance. Using one MPI process per node degrades performance for large matrices, and using one MPI process per CPU core degrades performance for small matrices. However, many legacy parallel applications only use MPI for parallelism, and parts of applications other than the solver often scale well in that case. Thus, we expect one process per CPU core to be a common configuration as well. Since we primarily intend TSQR as an orthogonalization kernel for iterative solvers, these two MPI configurations are the most important.

V. Accuracy

In this section, we show that TSQR produces results of comparable accuracy to the Householder QR factorization, as implemented sequentially in LAPACK and in parallel in ScaLAPACK. We show only a few illustrative results, but for both real and complex arithmetic. Our TSQR implementation in the Trilinos software library passes nightly accuracy tests.

TSQR promises results of similar accuracy as the Householder QR factorization, and potentially of greater accuracy than other orthogonalization methods in common use. TSQR's accuracy is why we favor it for Krylov methods, over the simpler and cheaper method of Stathopoulos and Wu [3]. (In Demmel et al. [1], we call their method "CholeskyQR." It

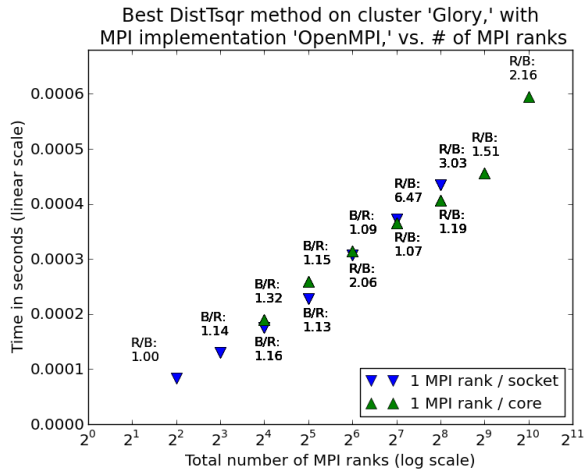

Fig. 8. Run time in seconds of `DistTsqr` on the Glory cluster, with OpenMPI 1.4.1 as the MPI implementation, for an "R stack" test problem. All other details are the same as Figure 7 (which see for a key).

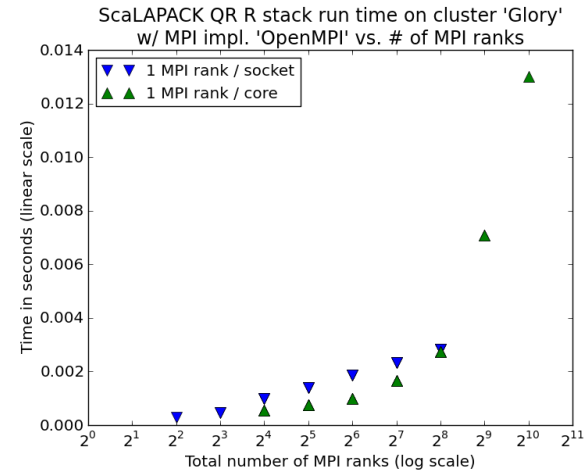

Fig. 9. Run time in seconds of `ScaLAPACK`'s `PDGEQRF` and `PDORMQR` routines in sequence, on the Glory cluster, with OpenMPI 1.4.1 as the MPI implementation, for the same "R stack" test problem used for Figures 7 and 8. As in Figures 7 and 8, blue downward-pointing triangles show results for 1 MPI rank per socket (4 ranks per node), and green upward-pointing triangles show results for 1 MPI rank per CPU core (16 ranks per node).

involves computing the Gram matrix with a single all-reduce operation, and solving the normal equations on the resulting small dense matrix.) What "accuracy" means for an application depends on what the orthogonal vectors and their coefficients are used to compute. For example, the Generalized Minimum Residual Method (GMRES) for solving $Ax = b$ uses the coefficients from the orthogonalization to solve a least-squares problem. However, a reasonable way to measure accuracy of a QR factorization independently of an application, is to look at the following two quantities:

- The normwise distance of the computed Q factor from an orthogonal matrix, for which a good approximation is $\|I - Q^*Q\|$
- The "residual" $\|A - QR\|$

We measure both of these quantities in the Frobenius norm $\|\cdot\|_F$. It turns out that computing a nearly orthogonal Q is harder for an orthogonalization method to achieve than a small residual norm. We compare the sequential and shared-memory parts of TSQR against LAPACK's QR factorization, and the distributed-memory part of TSQR against ScaLAPACK's QR factorization.

C++ data type	$\|I - Q^*Q\|_F$	$\|A - QR\|_F$	$\|A\|_F$
`float`	5.05527×10^{07}	2.14331×10^{06}	1.14881
`double`	1.74832×10^{15}	4.42286×10^{15}	1.1547
`complex<float>`	6.44408×10^{07}	2.19481×10^{04}	1.15019
`complex<double>`	1.73203×10^{15}	2.02212×10^{14}	1.1547

TABLE I
ACCURACY RESULTS FOR `TBBTSQR`, FOR BOTH REAL AND COMPLEX DATA, FOR BOTH 32-BIT IEEE-754 SINGLE PRECISION (`FLOAT`) AND 64-BIT IEEE-754 DOUBLE PRECISION (`DOUBLE`).

A. Sequential cache-blocked and shared-memory parallel TSQR

As in Section IV-C, we combine the sequential cache-blocked and shared-memory parallel parts of TSQR, since the former is just the latter running with only one thread. For a representative accuracy result, we tested `TbbTsqr` with 16 threads on a 1,000,000 by 10 test matrix, with a cache block size of 2^{16} bytes. The test matrix had a condition number of 10^{10}, with singular values 1, 0.1, 0.01, ..., 10^{-10}, and random orthogonal left and right singular vectors. We tested real and complex arithmetic, each with 32-bit and 64-bit IEEE 754 floating-point values (thus, the four data types supported by the BLAS and LAPACK). We show both a measure of orthogonality ($\|I - Q^*Q\|_F$) and a measure of the residual ($\|A - QR\|_F$). Table I shows the results. All results are close to machine epsilon, as LAPACK QR is as well. The residual for 32-bit complex floating-point numbers is a bit larger than expected; we are currently investigating whether this is of concern. Results for the other TSQR components are comparable, so we do not show them here.

VI. FULL TSQR PERFORMANCE

Here, we show performance results for the full TSQR implementation, composed of all the components described above. We compare it to straightforward implementations of Classical Gram-Schmidt (CGS) and Modified Gram-Schmidt (MGS) orthogonalization. TSQR is at least twice as fast as MGS and about 50% faster than CGS, and also scales well up to the maximum number of processors (16384) tested. We benchmark all these orthogonalizations in a complete software framework in the Trilinos library, which can be used today in block Krylov subspace methods. In Section VI-A, we summarize the software architecture of Trilinos' orthogonalizations. We compare the performance of TSQR, CGS, and MGS for a representative test problem in Section VI-B.

A. Orthogonalization in Trilinos

We have integrated TSQR into a full orthogonalization method in Trilinos' Anasazi and Belos iterative solvers packages. Baker et al. [21] give an overview of the software architecture of Anasazi, which provides iterative eigenvalue solvers. Belos, which provides solvers for linear systems, has a similar software architecture. Their design decouples the numerical algorithms from both distributed-memory and shared-memory parallelization. This makes it easy to compare TSQR fairly with other orthogonalizations. For example, all the vector operations in our MGS and CGS implementations use MPI and Intel's TBB for hybrid parallelism, just like TSQR. For details on Trilinos' shared-memory parallelization schemes, see Baker et al. [22].

Anasazi and Belos contain several different orthogonalization methods, with various performance and robustness features, including selective reorthogonalization. These methods all implement the `OrthoManager` and `MatOrthoManager` interfaces discussed in Baker et al. They operate on "multivectors," each of which is a block containing one or more vectors. Anasazi and Belos were designed to favor block iterative methods, though they do implement non-block iterations as well. As mentioned in Section I, operating on blocks instead of individual vectors can improve performance. Block methods can also improve accuracy when solving eigenvalue problems with clusters of eigenvalues. Anasazi's and Belos' orthogonalizations implement the "project" and "normalize" operations described in Section I. They also implement a combined "project and normalize" combined operation, that can save some work when projection and normalization follow in direct sequence. The algorithm used for projection for all experiments in this section, is for each block Q_j, first to compute $C_j := Q_j^* X$ via a single block reduction, and then to compute $X := X - Q_j C_j$ in parallel (which does not require communication). We call this "Block Modified Gram-Schmidt," since it projects one block at a time, just as MGS projects one vector at a time.

We have made our TSQR-based orthogonalization methods available in Anasazi and Belos, as `TsqrOrthoManager` (which implements the `OrthoManager` interface) and `TsqrMatOrthoManager` (which implements the `MatOrthoManager` interface). They pass nightly accuracy tests and can be used right now in iterative methods. Anasazi and Belos also allow orthogonalization with respect to an arbitrary inner product provided by an operator M, so that $\langle x, y \rangle = x^* M y$, where x^* denotes the (complex conjugate) transpose of x. TSQR itself can only orthogonalize with respect to the Euclidean inner product, but `TsqrMatOrthoManager` degrades to use a less efficient and less accurate orthogonalization method when a non-Euclidean inner product is required. For all tests in this paper, we use the standard Euclidean inner product $\langle x, y \rangle = x^* y$.

B. TSQR, MGS, and CGS performance

All the orthogonalization methods in Anasazi and Belos spend some of their time computing checks for whether reorthogonalization is necessary. These checks involve vector norms and can be expensive. Our TSQR-based method performs these checks also and has an option to reorthogonalize, but it does not compute the checks if reorthogonalization is not being performed. For a fair performance comparison, we implemented another simple orthogonalization method `SimpleOrthoManager`, that performs no reorthogonalization or checks for reorthogonalization. Its "project" phase uses simple Block Modified Gram-Schmidt, and its "normalize" phase can use either MGS or CGS. `SimpleOrthoManager` serves as a performance lower bound for the more accurate and robust schemes implemented in Anasazi and Belos. In the following tests, when we refer to "MGS" resp. "CGS," we mean `SimpleOrthoManager` set to use MGS resp. CGS for normalization. (Note that the projection phase uses *block* Gram-Schmidt, so what we call "MGS" should actually be faster than the textbook vector-by-vector MGS algorithm.)

As above, we performed all experiments on the "Glory" cluster. We ran with one MPI process per NUMA region, which as we mentioned in Section IV-D is an overall good choice for modern multicore architectures. On this architecture, this corresponds to with 4 MPI processes per node and 4 threads per MPI process. All three orthogonalizations tested use Intel's TBB library within an MPI process for shared-memory parallelism. We fixed the number of rows per node at 10^6, to measure weak scaling of the three methods, and fixed the number of columns per block at 10.

We performed two sets of experiments: normalization only (a single block), and a combination of normalization and projection (five blocks). Both sets include results for two different MPI implementations: MVAPICH 1.1, and OpenMPI 1.4.1. For all experiments, we first computed a single run, to remove the effects of the MPI library's performance tuning (which we found to be significant with OpenMPI) and the orthogonalization's own startup costs. We then timed 10 consecutive runs and computed the mean of the total time. Given the timer resolution of 10^{-6} seconds on this system, all timings in this section should be accurate to at least 3-4 decimal digits.

We expect that TSQR will consistently outperform both MGS and CGS by a constant factor (which is related to the number of columns) for all numbers of nodes. Run time for all three methods should only increase slightly with the number of nodes. This is verified by our performance results. Figures 10 and 11 show the single-block results for both MPI implementations (MVAPICH resp. OpenMPI), where we are only timing TSQR as a normalization. TSQR is consistently more than twice as fast as MGS and CGS for all numbers of nodes tested, up to the maximum number of nodes tested (1024 nodes, that is 16384 processors).

Figures 12 and 13 show the results for projection and normalization with 5 blocks of 10 columns each, for MVAPICH resp. OpenMPI. In this case, the normalization step (TSQR, CGS, or MGS) is performed 5 times, and the block projection operation $C_j := Q_j^* X$, $X := Q_j C_j$ is performed $1 + 2 + 3 + 4 = 10$ times. The latter is computed in the same

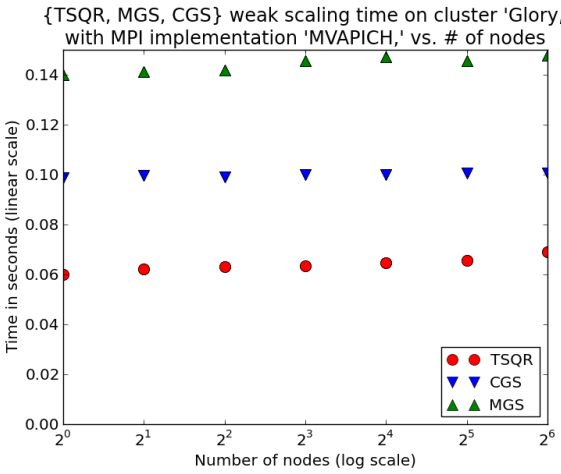

Fig. 10. Weak scaling test problem, exercising normalization only for a single multivector with 10 columns and 10^6 rows per node. The plot shows run time in seconds for TSQR, MGS, and CGS on various numbers of nodes on the Glory cluster, with MVAPICH 1.1 as the MPI implementation.

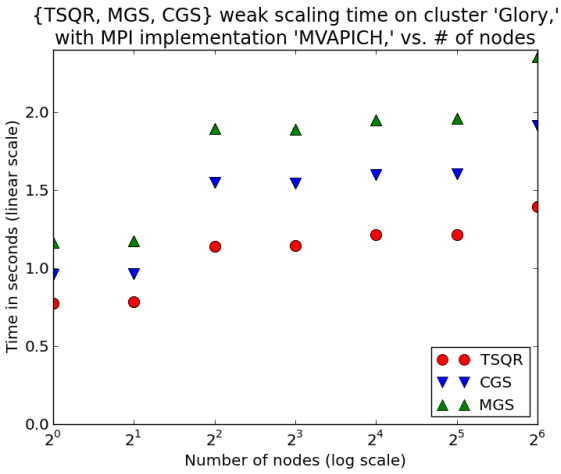

Fig. 12. Weak scaling test problem, exercising both projection and normalization for 5 multivectors with 10 columns each and 10^6 rows per node. The plot shows run time in seconds for TSQR, MGS, and CGS on various numbers of nodes on the Glory cluster, with MVAPICH 1.1 as the MPI implementation.

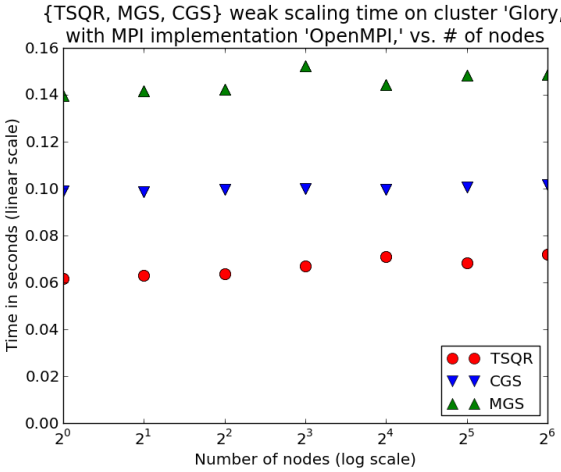

Fig. 11. Weak scaling test problem, exercising normalization only for a single multivector with 10 columns and 10^6 rows per node. The plot shows run time in seconds for TSQR, MGS, and CGS on various numbers of nodes on the Glory cluster, with OpenMPI 1.4.1 as the MPI implementation.

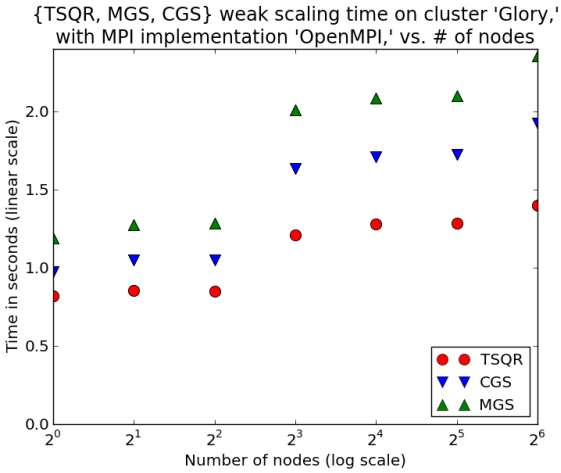

Fig. 13. Weak scaling test problem, exercising both projection and normalization for 5 multivectors with 10 columns each and 10^6 rows per node. The plot shows run time in seconds for TSQR, MGS, and CGS on various numbers of nodes on the Glory cluster, with OpenMPI 1.4.1 as the MPI implementation.

way for all three normalization schemes tested. Nevertheless, projection and normalization with TSQR was up to twice as fast as projection and normalization with CGS or MGS, on the maximum number of nodes tested (1024 nodes, that is 16384 processors). The jump in run times for four or more nodes must be due to the projection step, since we did not observe it in the experiments with only normalization. It has nothing to do with TSQR, MGS, or CGS, but it deserves further investigation.

VII. Conclusions and Ongoing Work

We have developed a TSQR framework in the Trilinos solvers library [23]. It has been available in standalone form since the 10.6 release of Trilinos, and will be available in the next 10.8 release for use as an orthogonalization in iterative methods. Our TSQR framework uses MPI for distributed-memory parallelism, and Intel's Threading Building Blocks (TBB) for shared-memory parallelism on a node. Its modular architecture makes it easy to port to different implementations of shared-memory parallelism. We are collaborating with Michael Anderson of the University of California Berkeley on a GPU module for this framework, which when completed will run TSQR on a cluster of GPUs. Combining this with Trilinos' other GPU capabilities will enable a new generation of iterative solvers that perform well on both CPU-only and hybrid CPU / GPU clusters. Michael Anderson's GPU TSQR will be integrated into the MAGMA project (see Agullo et al. [11]) for use as the panel factorization in a general QR factorization.

We also plan to develop and deploy communication-avoiding iterative solvers in Trilinos, such as those presented in [4] and [5]. This will require a hybrid-parallel implementation

of the matrix powers kernel (see Section I), which our collaborators are developing. As we discuss in [4], the performance of TSQR and this kernel can be linked, and "co-tuning" of the two kernels together at runtime may be necessary. Future work will explore runtime co-tuning of the matrix powers kernel and TSQR.

TSQR offers a fast and accurate orthogonalization method for small groups of vectors. Combined with Block Gram-Schmidt, it forms a complete orthogonalization solution specifically tuned for iterative methods. Our TSQR implementation exploits distributed-memory and shared-memory parallelism in a modular way, and works on both real and complex data of different precisions. It achieves significant speedups over competing orthogonalization methods, at scales from a single processor to at least 16,384 processors.

VIII. Acknowledgments

Many thanks to Chris Baker of Oak Ridge National Laboratory, and to Ross Bartlett, David Day, Mike Heroux, Rich Lehoucq, Heidi Thornquist, and many others of Sandia National Laboratories, for their input and many helpful discussions. Special thanks to the U.S. Department of Energy Office of Advanced Scientific Computing Research (ASCR), for funding this research through the Extreme-Scale Algorithms and Architectures Software Initiative (EASI) project. Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

References

[1] J. W. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR factorizations: theory and practice," University of California Berkeley, Tech. Rep. UCB/EECS-2008-89, 2008, also appears as LAPACK Working Note #204 (see http://www.netlib.org/lapack/lawns/downloads/).

[2] G. W. Stewart, "Block Krylov-Schur method, a block extension of a Krylov-Schur algorithm for large eigenproblems," *SIAM J. Matrix Anal. Appl.*, vol. 23, pp. 601–614, 2000.

[3] A. Stathopoulos and K. Wu, "A block orthogonalization procedure with constant synchronization requirements," *SIAM Journal on Scientific Computing*, vol. 23, no. 6, pp. 2165–2182, 2002.

[4] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, "Minimizing communication in sparse matrix solvers," in *Proceedings of the 2009 ACM/IEEE Conference on Supercomputing (New York, NY, USA)*, Nov 2009.

[5] M. Hoemmen, "Communication-avoiding Krylov subspace methods," Ph.D. dissertation, EECS Department, University of California Berkeley, 2010.

[6] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, "Avoiding communication in sparse matrix computations," in *IEEE International Parallel and Distributed Processing Symposium*, Apr 2008.

[7] ——, "Avoiding communication in computing Krylov subspaces," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2007-123, Oct 2007, available online at http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-123.html.

[8] G. H. Golub and R. R. Underwood, "The block Lanczos method for computing eigenvalues," in *Mathematical Software III*, J. R. Rice, Ed. New York, NY: Academic, 1977.

[9] D. P. O'Leary, "The block conjugate gradient algorithm and related methods," *Linear Algebra Appl.*, vol. 29, pp. 293–322, 1980.

[10] A. H. Baker, J. M. Dennis, and E. R. Jessup, "On improving linear solver performance: A block variant of GMRES," *SIAM J. Sci. Comp.*, vol. 27, no. 5, pp. 1608–1626, 2006.

[11] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects," *Journal of Physics: Conference Series*, vol. 180, 2009.

[12] G. W. Stewart, "Block Gram-Schmidt orthogonalization," *SIAM J. Sci. Comput.*, vol. 31, no. 1, pp. 761–775, 2008.

[13] N. J. Higham, "Computing the polar decomposition with applications," *SIAM J. Sci. Stat. Comput.*, vol. 7, no. 4, pp. 1160–1174, 1986.

[14] J. W. Demmel, M. Hoemmen, Y. Hida, and E. J. Riedy, "Non-negative diagonals and high performance on low-profile matrices from Householder QR," University of California Berkeley, Tech. Rep. UCB/EECS-2008-76, May 2008, also appears as LAPACK Working Note #203 (see http://www.netlib.org/lapack/lawns/downloads/).

[15] E. J. Candès, X. Li, Y. Ma, and J. Wright, "Robust Principle Component Analysis," Department of Statistics, Stanford University, Tech. Rep. 2009-13, 2009.

[16] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 2.2*. High Performance Computing Center Stuttgart (HLRS), 2009.

[17] J. Reinders, *Intel Threading Building Blocks*. O'Reilly, 2007.

[18] H. C. Edwards, "Trilinos ThreadPool Library v1.1," Sandia National Laboratories, Tech. Rep. SAND2009-8196, 2009.

[19] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[20] Network-Based Computing Laboratory, "MVAPICH: MPI over Infiniband, 10GigE/iWARP and RoCE," available online at http://mvapich.cse.ohio-state.edu [last accessed Jan 2011].

[21] C. G. Baker, U. L. Hetmaniuk, R. B. Lehoucq, and H. K. Thornquist, "Anasazi software for the numerical solution of large-scale eigenvalue problems," *ACM Trans. Math. Softw.*, vol. 36, no. 3, July 2009.

[22] C. G. Baker, H. C. Edwards, M. A. Heroux, and A. B. Williams, "A lightweight API for portable multicore programming," in *18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2010)*. IEEE, 2010.

[23] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the Trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, 2005.